An IDE for the Design, Verification and Implementation of Security Protocols

Rémi Garcia
Département Informatique
IUT de Bordeaux
Bordeaux, France
Email: remi.garcia@etu.u-bordeaux.fr

Paolo Modesti
Faculty of Computer Science
University of Sunderland
Sunderland, United Kingdom
Email: paolo.modesti@sunderland.ac.uk

***Abstract*—Security protocols are critical components for the construction of secure and dependable distributed applications, but their implementation is challenging and error prone. Therefore, tools for formal modelling and analysis of security protocols can be potentially very useful to support software engineers. However, despite such tools having been available for a long time, their adoption outside the research community has been very limited. In fact, most practitioners find such applications too complex and hardly usable for their daily work. In this paper, we present an Integrated Development Environment for the design, verification and implementation of security protocols, aimed at lowering the adoption barrier of formal methods tools for security. In the spirit of Model Driven Development, the environment supports the user in the specification of the model using the simple and intuitive language AnB (and its extension AnBx). Moreover, it provides a push-button solution for the formal verification of the abstract and concrete models, and for the automatic generation of Java implementation. This Eclipse-based IDE leverages on existing languages and tools for the modelling and verification of security protocols, such as the AnBx Compiler and Code Generator, the model checker OFMC and the cryptographic protocol verifier ProVerif.**

***Keywords*-Security Protocols; Design; Verification; Implementation; Integrated Development Environment;**

I. Introduction

The ubiquitous usage of information and communication technologies offers individuals and organisations an enormous number of opportunities for business and social interaction. However, it also poses significant risks and threats since vulnerabilities can be exploited by attackers to gain access to confidential data, and compromise the integrity of connected systems. Many experts agree that the root cause of vulnerabilities is incorrect software [1]. Security protocols play a key role in protecting user data exchanged over a network infrastructure that can be assumed to be under adversary control, as in the Dolev-Yao attacker model [2]. However, programming security protocols is challenging and error-prone, as experience has shown that low-level implementation bugs are discovered even in protocols like TLS and SSH, which are widely used and thoroughly tested. Therefore, tools for formal modelling and analysis of security protocols can be very useful to support software engineers. Formal specification helps to better understand system requirements, and a formal model, suitable for automatic analysis, can detect inconsistencies and requirements errors at an early stage of the development. It is also cost effective as errors discovered at later stages are generally more expensive to fix [3].

However, security requirements are particularly challenging because they need to consider the behaviour of an active adversary. To help reasoning about the security properties, the specification of security protocols with high-level programming abstractions, suited for security analysis and automated verification, has been advocated by the formal methods for security research community [4], [5]. This was also one of the reasons for developing tools for the verification of security protocols in the symbolic model [6], [7], [8], and for the automatic generation of security protocols implementations [9], [10], [11]. Despite such tools having been available for a long time, their adoption outside the research community has been very limited. In fact, most practitioners find this kind of applications too complex and hardly usable, including the difficulty to write and understand the formal specification. For these reasons, they are reluctant to use such tools for their daily work. In order to lower the adoption barrier we advocate an approach based on:

- a simple and intuitive language for the formal specification of security protocols;
- a Model-Driven Development (MDD) strategy allowing automatic generation of a program, from a simple and abstract model that can be formally verified;
- an Integrated Development Environment supporting the developer.

To demonstrate this approach, we present the *AnBx*-IDE[1], an Integrated Development Environment for the design, verification and implementation of security protocols. This Eclipse-based IDE leverages on existing languages and tools for modelling and verification of security protocols, such as the *AnBx Compiler and Code Generator* [10], for the automatic generation of Java implementations from a model described in the simple Alice & Bob (*AnB*) notation [12] (or its extension *AnBx* [13]), and, for the automated

[1]Available at http://www.dais.unive.it/~modesti/anbx/ide/

```
Protocol: Example_AnBx
Types:
    Agent A,B;
    Certified A,B;
    Number Msg;
    SymmetricKey K;
    Function [Agent,Number -> Number] log
Knowledge:
    A: A,B,log;
    B: B,A,log
Actions:
    A -> B, @(A|B|B): K
    B -> A: {|Msg|}K
    A -> B: {|hash(Msg),log(A,Msg)|}K
Goals:
    K secret between A,B
    Msg secret between A,B
    A authenticates B on Msg
    B authenticates A on K
    B authenticates A on Msg
```

Figure 1. *AnBx* Protocol Example

verification, the symbolic model checker *OFMC* [7] and the cryptographic protocols verifier *ProVerif* [6].

The IDE, along with the interaction with the back-end tools, includes many features meant to help programmers to increase their productivity, like syntax highlighting, code completion, code navigation and quick fixes.

The component responsible for the integration with Eclipse [14] is a plug-in developed with XText [15], [16], a framework for the design and implementation of Domain Specific Languages (DSL). Given that the *AnBx* tool generates Java code, the choice of Eclipse, one of the most popular IDEs among Java professional developers, is meant to simplify the adoption among users familiar with it, but we believe our approach is general enough to be applicable also to other environments.

The outline of the paper is the following: in section II we introduce the specification language and the back-end tools. In section III we present the IDE construction and features. Finally, in sections IV and V we report about our preliminary evaluation and discuss the related and future work.

II. Specification Language and Back-End Tools

The IDE leverages on existing languages and tools for modelling and verification of security protocols, such as the AnBx Compiler and Code Generator, the model checker OFMC and the cryptographic protocol verifier ProVerif. In the spirit of MDD, the environment supports the user in the specification of the model using the simple and intuitive language *AnB* (and its extension *AnBx*). Moreover, it provides a push-button solution for the formal verification of abstract and concrete model, and for the automatic generation of Java implementation.

```
Protocol: Example AnB
Types:
    Agent A,B;
    Number Msg,Nonce;
    SymmetricKey K;
    Function pk,sk,hash;
    Function log
Knowledge:
    A: A,B,pk,sk,inv(pk(A)),inv(sk(A)),hash,log;
    B: A,B,pk,sk,inv(pk(B)),inv(sk(B)),hash,log
Actions:
    A -> B: A
    B -> A: {Nonce,B}pk(A)
    A -> B: {{Nonce,B,K}inv(sk(A))}pk(B)
    B -> A: {|Msg|}K
    A -> B: {|hash(Msg),log(A,Msg)|}K
Goals:
    K secret between A,B
    Msg secret between A,B
    A authenticates B on Msg
    B authenticates A on K
    B authenticates A on Msg
    inv(pk(A)) secret between A
    inv(sk(A)) secret between A
```

Figure 2. *AnB* Protocol Example

A. AnBx Language

The *AnBx* language is formally defined in [13] and is built as an extension of *AnB* [12]. The main peculiarity of *AnBx* is to use channels as the main abstraction for communication, providing different authenticity and confidentiality guarantees for message transmission. The translation from *AnBx* to *AnB*, can be parametrised using different channel implementations, by means of different cryptographic operations.

Figure 1 shows an example protocol in which two agents want to exchange securely a message `Msg`, using a freshly generated symmetric key `K`, i.e. a key that is different for each protocol run. If `K` is compromised, neither previous nor subsequent message exchanges will be compromised, but only the current one. This is similar to what happens in TLS were a symmetric session key is established (using asymmetric encryption) at the beginning of the exchange. It should be also noted that this setting is also more efficient, as symmetric encryption is notoriously much faster than the asymmetric one. Therefore, if the size of the message is significant, using symmetric encryption should be preferable.

The `Types` section includes declaration of identifiers of different types, and functions declaration, while the section `Knowledge` denotes the initial knowledge of each agent. An optional section, `Definitions`, can be used to specify macros with parameters. In the `Actions` section, the action `A -> B,@(A|B|B):K` means that the key `K` is generated by `A` and sent on a secure channel to `B`. The notation `@(A|B|B)` denotes the properties of the channel: the message exchanged originates from `A`, it is freshly generated (`@`), verifiable by `B`, and secret for `B`. How the channel is implemented is delegated to the compiler. The

designer can select between different options, or simply use the default one. This simplifies the life of the designer, who does not need to be in charge of low level implementation details. A translation to *AnB* is shown in Figure 2: in this case the channel is implemented using a challenge-response technique, where `B` freshly generates a `Nonce` (the challenge), encrypted with `pk(A)`, the public key of `A` along with the sender name (`{.}` denotes the asymmetric encryption). This guarantees that only `A` would be able to decrypt the incoming message.

The response, along with the challenge, includes the symmetric key `K`. The response is digitally signed with `inv(sk(A))`, the private key of `A` and then encrypted with `pk(B)`, the public key of `B`. This will allow `B` to verify the origin of the message and that `K` is known only by `A` and `B`.

It should be noted, that in *AnBx* we abstract from these cryptographic details and we simply denote the capacity of `A` and `B` to encrypt and digitally sign using a Public Key Infrastructure (PKI) with the keyword `Certified`. This reflects the customary practice of a Certification Authority to endorse public keys of agents, usually issuing X.509 certificates, allowing every agent to verify the identity associated with a specific public key. Moreover, in *AnBx*, keys for encryption and for signature are distinguished by using two different symbolic functions, `pk` and `sk` respectively.

Once the symmetric key `K` is shared securely between `A` and `B`, then `B` can send the payload `Msg` secretly (`{|.|}` denotes the symmetric encryption). Finally, `A` acknowledges receipt, replying with a digest of the payload computed with the `hash` function (a predefined function available in *AnBx*), and with a value computed with the `log` function.

The section `Goals` denotes the security properties that the protocol is meant to convey. They can also be translated into low level goals suitable for verification with various tools. Supported goals are: 1) *Weak Authentication* goals have the form `B weakly authenticates A on Msg` and are defined in terms of non-injective agreement [17]; 2) *Authentication* goals have the form `B authenticates A on Msg` and are defined in terms of injective agreement on the runs of the protocol, assessing the freshness of the exchange; 3) *Secrecy* goals have the form `Msg secret between A1,...,An` and are intended to specify which agents are entitled to learn the message `Msg` at the end of a protocol run.

In the example protocol (Figure 1), the desirable goals are the secrecy of the symmetric key `K` and of the payload `Msg` that should be known only by `A` and `B`. There are also authentications goals. `B` should be able to verify that `K` originates from `A` and the key is freshly generated. Finally, two goals express the mutual authentication between `A` and `B` regarding `Msg`, including the freshness of the message. In summary, this protocol allows two agents to securely exchange a message, with guarantees about the origin and the freshness of the message.

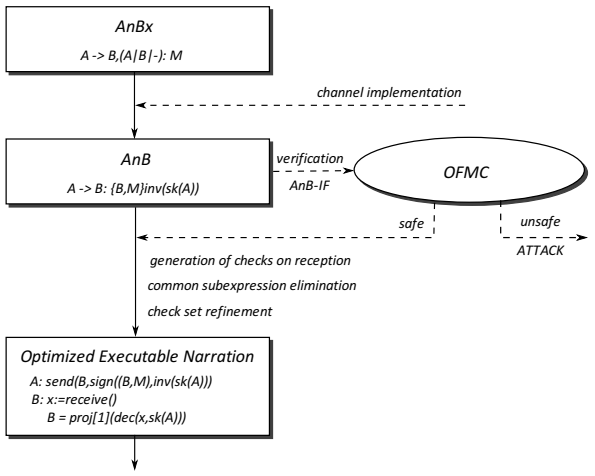

Figure 3. Compiler front-end: pre-processing, verification, *ExecNarr* optimization

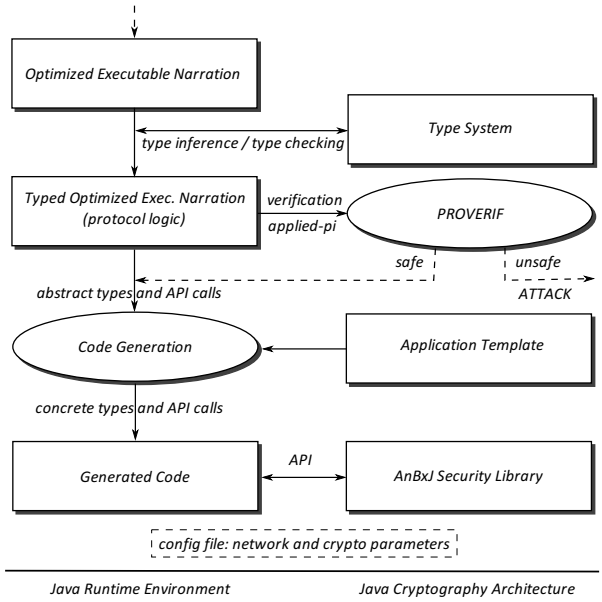

Figure 4. Compiler back-end (Type System, Code generator, Verification) and run-time support

B. AnBx Compiler and Code Generator

We briefly present the *AnBx* Compiler and Code Generator by illustrating the main steps in the automatic Java code generation of security protocols from an *AnBx* or *AnB* model (Figure 3). A more detailed description is available in [10].

The input protocol is lexed, parsed and then compiled to *AnB*, a format allowing the automated verification with the OFMC model checker. If the verification is successful, the *AnB* specification can be compiled into an *executable narration* (*ExecNarr*), a sequence of actions that operationally encodes how agents are expected to execute

the protocol. In particular, are computed the checks on receptions, actions that agents have to perform on incoming messages, in order to verify that the protocol is running according to the specification. For example, verification of digital signatures, decryption of incoming messages, equality tests in order to check if the incoming message matches the prior knowledge. Then, the sequence of actions can be reordered (*optimized executable narration* (*Opt-ExecNarr*) [18]) applying optimization techniques like the common sub-expression elimination (CSE) in order to minimise the number of cryptographic operations and reduce the overall execution time.

The generation of the Java code requires a further step (Figure 4). We model the *protocol logic* by means of a language independent intermediate format called *Typed-Opt-ExecNarr*, which is, in essence, a typed representation of the *Opt-ExecNarr*, useful to parametrize the translation and to simplify the emission of code in the concrete programming language. The type system infers the type of expressions and variables ensuring that the generated code is well-typed. It has the additional benefit of detecting at run-time whether the structure of the incoming messages is equal to the expected one, according to the protocol specification.

The Java code emission is done by instantiating the protocol templates (*application logic*), i.e., the skeleton of the application, using the information derived from the protocol logic. It is worth noting that only at this final stage the language specific features and their API calls are actually bound to the protocol logic. The verification of the protocol logic can be done with ProVerif, translating from the *Typed-Opt-ExecNarr* to the Applied pi-calculus. [6].

In summary, ProVerif verifies the symbolic implementation from which the Java code is emitted, while OFMC verifies the abstract model edited by the designer.

C. OFMC

OFMC [7] uses the AVISPA Intermediate Format IF [19] as "native" input language. IF allows to describe security protocols as an infinite-state transition system using set-rewriting. OFMC also supports the *AnB* language. OFMC performs both protocol falsification and bounded session verification by exploring, in a demand-driven way, the transition system. The major techniques employed by OFMC are the lazy intruder, which is a symbolic representation of the intruder, and constraint differentiation, which is a search-reduction technique that integrates the lazy intruder with ideas from partial-order reduction achieving a reduction of the search space associated without excluding attacks (or introducing new ones).

D. ProVerif

ProVerif [6] is an automated verifier for cryptographic protocols, modelling the protocol and the attacker according to the Dolev-Yao [2] symbolic approach, which in essence represents data and ideal cryptographic operations symbolically, assuming the attacker has complete control over public communication channels. Differently from model checkers, ProVerif can model and analyse an unbounded number of parallel sessions of the protocol. However, like model checkers, ProVerif can reconstruct a possible attack trace when it detects a violation of the intended security properties. ProVerif may report false attacks, but if a security property is reported as satisfied then this is true in all cases, so it is necessary to analyse the results carefully when attacks are reported.

III. Integrated Development Environment

A. Motivation

Despite not being strictly necessary to build programs, Integrated Development Environments are not only useful to increase productivity but also to lower the adoption barrier for methodologies and technologies. In our case, while working independently with the back-end tools is fine for experts, our experience shows that new users, unfamiliar with formal methods, languages and tools for security, may face some difficulties. In particular, we rely on the second author's experience acquired supporting researchers using such tools, and reflections made observing 3rd-year undergraduate students at the University of Sunderland working during tutorials on programming security protocols, where some of these tools where employed.

Nowadays, most programmers expect development tools to be supported by IDEs. Therefore, their integration, within an environment they are familiar with, can make developers more open and interested to adopt such tools. An IDE can also greatly simplify the setup and configuration of the environment, an issue that could deter new users.

There are also challenges for learners to deal with security concepts. To counter this problem, an intuitive language may help, but despite being considerably simpler than other languages (e.g. the applied-pi calculus used by ProVerif), *AnBx*/*AnB* can still present technical challenges to beginners. Sometimes the difference between symmetric and asymmetric encryption is not immediately understood. Additionally, for asymmetric encryption, the usage of keys for different purposes (i.e. encryption and signature) is crucial as key material does not need to be confused [20]. Since *AnB* lacks of function type signature, we prefer *AnBx* to allow the IDE to check the arity and type of arguments while editing. Regarding *AnBx*, the channel syntax (see Figure 1) is compact and effective, but first time users need to assimilate it. The IDE supports the user to overcome such difficulties.

In a nutshell, the main benefits of our IDE are:

- an integrated environment for the modelling, analysis, and implementation of security protocols in an MDD context;
- a simple intuitive input language (with syntactic support, type checking, etc.);

- push-button tool that integrates existing tools and simplifies their usage;
- enabling a real-time modelling/verification feedback cycle which is crucial to increase productivity.

B. XText

In order to support the *AnBx* modelling and the integration of Eclipse with the back-end tools, we used Xtext [15], [16], a standard framework for the development of programming languages and domain-specific languages in the Eclipse ecosystem, supported by an active community. It also allows integration with IntelliJ and web browsers. By defining the grammar of the language in the Extended Backus-Naur Form (EBNF), Xtext provides a set of features including parser, linker, type-checker, compiler as well as editing support for Eclipse: handling of cross-references, code completion, navigation, syntax colouring, validation and more. While the default behaviour of Xtext is optimized to cover a wide range of languages and use cases, every language is different and consequently, also for *AnBx*, we needed to customize the behaviour of various features, notably type-checking and quick fixes provider.

C. Editing a project

In the rest of this section we refer to our example protocol in Figure 1 and provide some examples of tool support for editing, verifying, and running the protocols.

Before starting to write code, the user needs to run a wizard, allowing to create an *AnBx* project with a few clicks, by means of the `File→New→Project` menu commands. A stub is generated, similar to the "Hello world" code in others languages. Another wizard allows to create additional *AnBx* files.

The editor supports the user in many ways. As mentioned, by defining the grammar of the *AnBx* language, plus some customisations, Xtext provides a fully fledged editor. We exemplify here some features that are useful to avoid mistakes in writing protocol specifications.

1. Context-dependent variables are a useful feature, for the definitions of macros with parameters. Parameters allow the user to produce a generic macro and call it with concrete expressions as parameters, maximizing the code's flexibility. For example, in Figure 5, the `Z` parameter of `Def1` is not available outside its declaration.

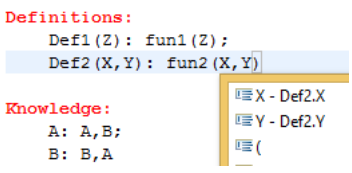

Figure 5. Local variables' UI consequences

2. The editor can help to avoid possible mistakes with the *AnBx* channel's syntax, a triple (*auth*|*vers*|*dest*). For example,

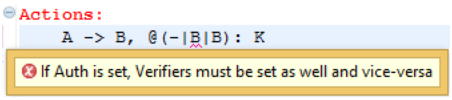

Figure 6. Auth and Verifiers checking

the *auth* parameter (the agent authenticating a message, e.g., with a signature) and the *vers* parameter (a set of agents, that the *auth* intends to be able to verify the authenticity of the message) can occur in a single channel mode only together, but not alone (Figure 6).

3. The protocol in Figure 7 has two errors. The first one is due to the fact that the $auth$ parameter of the channel mode must be of type `Agent`. The type checker returns the error cause and provides two suggestions on how to resolve it. The second error occurs because some cryptographic notations are not legal. In fact, a challenge for a developer is to figure out which encryption to use, in which situation. The validation system guarantees the correctness of the types, according to the cryptographic specification. The user is trying to use a public key in a symmetric cipher scheme. To fix the error the user has two possibilities: changing the type of `K` to `SymmetricKey` or changing the encryption scheme from symmetric to asymmetric. However, only in the first case the protocol is safe, while in the second case there is an attack. This can be determined verifying the protocol.

These three examples show how the IDE can help users unfamiliar with cryptography to avoid design mistakes.

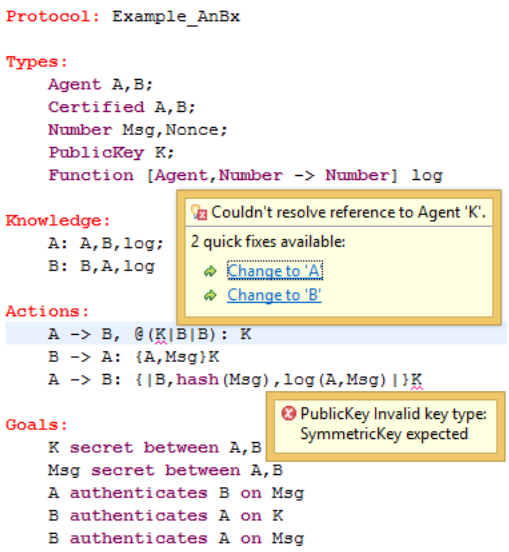

Figure 7. Error message and quick fix

Validation: In general, the IDE enforces several sanity checks and signals their violation to the user. For example, double declarations are forbidden and validation allows us to check them. Moreover, the validation feature notably handles arity of function calls and type-checking. For example. a

function signature like `log : Agent, Number → Number` will be compared to its calls and will accept only `Agent` or function returning an agent as first parameter. In the same way, a `log` call can not be used as a parameter except if the other signature expects a `Number`.

Quick fixes: Quick fixes are handled by Xtext when the grammar specifies a strict requirement. For example, with the channels notation, only a declared agent can be inside the notation. Accordingly, a tooltip window appears to indicate possible fixes as shown in Figure 7. While some fixes are inferred from the grammar specification, others require a specific customisation of the quick fix rules.

D. Integration with tools

1) Running tools: The integration of verification tools allows the user to run common verification and code generation tasks with a few clicks. For example, the "Launch associated validator" option triggers automatic verification with OFMC or ProVerif, depending on the chosen output format. It should be noted that enabling a real-time modelling/verification feedback cycle is crucial to increase productivity when dealing with complex protocols.

As the *AnBx* compiler allows users to generate a Java implementation for the specified protocol, the code can automatically run as soon as the code is generated. Alternatively, the associated Java project can be opened in Eclipse and built through an Ant file, a standard build file for the Eclipse platform and other IDEs.

Thus, we cover the entire process of writing, running and generating code within the same environment. Combining a simple language with a simple interface enables the developer to code and verify any protocol in a user-friendly way, providing the necessary functions for a complete MDD workflow.

2) Configuration of the cryptographic engine: Another significant feature is the possibility to change the cryptographic engine settings allowing to use different cipher schemes and parameters without the need to regenerate or recompile the application but simply editing the configuration file. This can be useful when it is necessary to increase the level of security of the cryptographic schemes in use.

IV. Evaluation and Related Work

1) Evaluation: The plug-in is platform-independent and it has been tested on Windows, Linux and Mac. Moreover, the full protocol suite available with the *AnBx* compiler package (50 protocols) has been successfully validated. Despite being made available to users only very recently, the IDE has been already used satisfactory by some security researchers and post-graduate research students at the universities of Newcastle and Sunderland. So far, their work involves mostly modelling and verification of e-payment protocols. While we had not performed yet a formal user evaluation, a general appreciation was given to the simplicity of the specification language and the user-friendliness of the IDE. Anecdotally, we can report that a security researcher stated that finally he could write his models with a language he understands using the familiar Eclipse environment.

2) Related work: We share with the SPaCIoS tool [21] the idea of providing support for the automatic verification and validation of security properties, leveraging on existing tools. However, while their range of supported tools is wider than ours, it lacks (as it is left for a future work) a simpler modelling language like the Alice & Bob notation. The authors acknowledge that overcoming this limitation would make the tool more suitable for practitioners. Moreover, SpaCIoS focuses on the analysis and not on MDD.

SPI2JavaGUI [22]), along with the protocol verification done by ProVerif, generates Java code from a graphical model based on the Spi calculus. We share the MDD approach, but our specification language is only textual. Assessing which of the two approaches is more intuitive and effective would require a specific investigation, but protocol narrations like *AnB* are generally considered more intuitive than process calculi.

Regarding the implementation of Eclipse-IDEs, [23] incorporates semantic verification techniques for DSLs defined in Xtext. While Xtext offers good support for the DSL's syntax, semantics support has been rather neglected. As a proof of concept this work considers a simple State-Transition-DSL and the editor verifies on the fly, using theorem prover Princess, that the model holds some semantic properties. In our case, the editor offers a limited semantics support, but all semantic features are available through to the back-end tools, the *AnBx* compiler in particular. The current limitation is that the user has to rely on the messages displayed on the Eclipse console.

[24] proposes a general approach for integration of verification tools and IDEs. It focuses on the semi-automatic verification tool KeY [25] for the Java language, with the aim to keep implementation, specification and proofs in sync. With respect to our IDE, a significant difference is that here the developer works on the concrete Java code, while we work on the abstract model which is remarkably simpler than the implementation. In our case, however, we face some challenges and limitations because of the gap between the formal model and the concrete verified model. For example, OFMC translates *AnB* to IF, and checks the IF state transition system, and see if an attack state can be reached. OFMC is able to tell which kind of property is violated (Authentication, Weak Authentication, Secrecy), providing an attack trace, but does not tell which goal is violated. Therefore, it is not easy to backtrack to the abstract model and inform the user about the specific violated goal. Instead, ProVerif provides information about every goal, and in the future we could annotate the *AnBx* compiler translation process to be able to trace back the tool's messages from the concrete to the abstract model.

V. Future work and Conclusion

We think that our IDE for the design, verification and implementation of security protocols may be of interest, not only to security researchers, but also to practitioners. We also believe that the approach is general enough to be applicable to other IDE platforms. In order to support new languages, our IDE can be extended but the amount of work will depend on the specific nature of the language. New verification tools could be also added either by extending our back-end tool (*AnBx* compiler) or integrating directly an existing compiler supporting the verification tool (similarly to what done for OFMC and ProVerif). However, a significant challenge will be proving the correctness of the translation between different formats.

We plan to overcome some of the limitations discussed above, in particular, working on the interpretation of analysis results, having a further integration with the verification tools and backtracking error messages to the source code of the model. Moreover, we could run a formal evaluation of the user experience with the IDE and extend the range of supported tools and input languages.

Acknowledgement

Rémi Garcia was supported by the EU Erasmus Programme during his visit at the University of Sunderland. The authors would like to thank Leo Freitas and the anonymous reviewers for their constructive comments.

References

[1] M. Dark, S. Belcher, M. Bishop, and I. Ngambeki, "Practice, practice, practice... secure programmer!" in *Proceeding of the 19th Colloquium for Inf. System Security Education*, 2015.

[2] D. Dolev and A. Yao, "On the security of public-key protocols," *IEEE Transactions on information Theory*, vol. 2, no. 29, 1983.

[3] I. Sommerville, *Software Engineering, 9th edition*. Addison-Wesley, 2010.

[4] M. Bugliesi and R. Focardi, "Language based secure communication," in *Computer Security Foundations Symposium, 2008. CSF'08. IEEE 21st*, 2008, pp. 3–16.

[5] M. Avalle, A. Pironti, and R. Sisto, "Formal verification of security protocol implementations: a survey," *Formal Aspects of Computing*, vol. 26, no. 1, pp. 99–123, 2014.

[6] B. Blanchet, "An efficient cryptographic protocol verifier based on Prolog rules," in *Computer Security Foundations Workshop, IEEE*. IEEE Computer Society, 2001.

[7] D. Basin, S. Mödersheim, and L. Viganò, "OFMC: A symbolic model checker for security protocols," *International Journal of Inf. Security*, vol. 4, no. 3, pp. 181–208, 2005.

[8] S. Meier, B. Schmidt, C. Cremers, and D. Basin, "The tamarin prover for the symbolic analysis of security protocols," in *International Conference on Computer Aided Verification*. Springer, 2013, pp. 696–701.

[9] M. Avalle, A. Pironti, D. Pozza, and R. Sisto, "JavaSPI: A framework for security protocol implementation," *International Journal of Secure Software Engineering*, vol. 2, no. 4, pp. 34–48, 2011.

[10] P. Modesti, "AnBx: Automatic generation and verification of security protocols implementations," in *8th International Symposium on Foundations & Practice of Security*, ser. LNCS, vol. 9482. Springer, 2015.

[11] O. Almousa, S. Mödersheim, and L. Viganò, "Alice and Bob: Reconciling formal models and implementation," in *Programming Languages with Applications to Biology and Security*, ser. LNCS. Springer, 2015, vol. 9465, pp. 66–85.

[12] S. Mödersheim, "Algebraic properties in Alice and Bob notation," in *International Conference on Availability, Reliability and Security (ARES 2009)*, 2009, pp. 433–440.

[13] M. Bugliesi, S. Calzavara, S. Mödersheim, and P. Modesti, "Security protocol specification and verification with AnBx," *Journal of Information Security and Applications*, vol. 30, pp. 46–63, 2016.

[14] Eclipse Foundation, "Eclipse IDE," http://www.eclipse.org.

[15] L. Bettini, *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[16] Eclipse Community, "Xtext documentation," 2017, http://eclipse.org/Xtext/documentation/.

[17] G. Lowe, "A hierarchy of authentication specifications," in *CSFW'97*. IEEE Computer Society Press, 1997, pp. 31–43.

[18] P. Modesti, "Efficient Java code generation of security protocols specified in AnB/AnBx," in *Security and Trust Management, STM 2014, Proceedings*, 2014, pp. 204–208.

[19] AVISPA, "Deliverable 2.3: The Intermediate Format," 2003, available at www.avispa-project.org.

[20] M. Abadi and R. Needham, "Prudent engineering practice for cryptographic protocols," in *IEEE Computer Society Symposium on Research in Security and Privacy*, 1994, pp. 122–136.

[21] G. Pellegrino, L. Compagna, and T. Morreggia, "A tool for supporting developers in analyzing the security of web-based security protocols," in *IFIP International Conference on Testing Software and Systems*. Springer, 2013.

[22] P. B. Copet, A. Pironti, D. Pozza, R. Sisto, and P. Vivoli, "Visual model-driven design, verification and implementation of security protocols," in *High-Assurance Systems Engineering (HASE),14th International Symposium on*. IEEE, 2012.

[23] T. Baar, "Verification support for a state-transition-dsl defined with Xtext," *Perspectives of System Informatics*, Jan. 2016.

[24] M. Hentschel, S. Käsdorf, R. Hähnle, and R. Bubel, "An interactive verification tool meets an ide," in *International Conference on Integrated Formal Methods*. Springer, 2014.

[25] B. Beckert, R. Hähnle, and P. H. Schmitt, *Verification of object-oriented software: The KeY approach*. Springer, 2007.